Out of Core, Out of Mind: Practical Parallel I/O

D. E. Womble, D. S. Greenberg, R. E. Riesen, S. R. Wheat

Sandia National Laboratories
Albuquerque, NM 87185–5800

(Proceedings of the Scalable Libraries Conference, October 6–8, 1993, Mississippi State University, pp. 10–16.)

Abstract

Parallel computers are becoming more powerful and more complex in response to the demand for computing power by scientists and engineers. Inevitably, new and more complex I/O systems will be developed for these systems. In particular we believe that the I/O system must provide the programmer with the ability to explicitly manage storage (despite the trend toward complex parallel file systems and caching schemes). One method of doing so is to have a partitioned secondary storage in which each processor owns a logical disk. Along with operating system enhancements which allow overheads such as buffer copying to be avoided and libraries to support optimal remapping of data, this sort of I/O system meets the needs of high performance computing.

1 Introduction

The solution of Grand Challenge Problems will require computations which are too large to fit in the memories of even the largest machines. The speed of individual processors is growing too fast to be matched with increased memory size economically. Successful high performance programs will have to be designed to run in the presence of a memory hierarchy. Great efforts have already been made to optimize computations for the fastest end of the hierarchy – the use of high speed registers and caches. The result has been the creation of optimized codes such as those in the BLAS. At least as large an effort must now be made to address the slow end of the hierarchy – the use of secondary storage such as disks or even SRAM backing stores.

The term I/O typically refers to the slow end of the spectrum and often involves the transfer of data between main memory and a disk system or between machines. I/O is often left in the control of the operating system (OS) such as in file systems and virtual memory systems. While these systems may be easy to use, their general nature makes them inefficient for many specialized scientific applications. In these cases, the I/O must be explicitly managed by the applications progammer either explicitly, or though calls to libraries. In the future, programmers may be able accomplish efficient I/O by providing "hints" to I/O management systems in the operating system.

In this paper, we will describe our experiences in producing high performance scientific codes that require the use of disks for temporary storage of data. We will discuss both algorithmic issues (e.g., ways to structure the code to reduce the I/O bottleneck) and systems issues (e.g., features of the OS that can make it easier to produce high performance codes). To illustrate these issues, we will use LU factorization for dense matrices. LU factorization is a common scientific kernel used in the solution of linear systems and the data movement is representative of many dense matrix algorithms.

2 Characteristics of I/O

I/O can generally be divided into three phases: the initial input of data, the maintenance of temporary data, and the output of program results.

The initial input of data often involves the transfer of data over HiPPI, from disk, or from another computer. It is a one time operation, and the limiting factor is often the speed of the hardware. Although it can lead to significant overhead, it is rarely an insurmountable bottleneck. An important issue is the format of the data. Because the input to one program may be the output to another, data may not be input in the most efficient format. Thus, the translation of data from one format to another is an important operation. We will have more to say about this later.

The output phase has similar characteristics to input phase. As before, a translation of the data may be necessary, such as in the display of data.

The handling of temporary values is much more problematic. Temporary values must be both written and read, and the order in which they are accessed can change over time. If the management of temporary memory is not efficient, it can slow the whole computation to a crawl. Virtual memory is one possible approach to maintaining temporary storage which is being provided by vendors. While this simplifies the programming, no virtual memory system can perform as well as a code written by a programmer who understands the algorithm being implemented; the overhead of a virtual memory system often defeats the advantage of using a parallel supercomputer, i.e., computational speed.

3 Systems for I/O

Our goal is to achieve high performance using large numbers of processors, and we cannot afford to pay the overhead of a general purpose memory management system such as virtual memory. Instead, we will focus on the explicit management of temporary data by a programmer who understands the algorithm and can tailor the I/O to the algorithm.

There are two approaches to I/O supporting this type of use, the parallel file system (PFS) and what we call partitioned secondary storage (PSS). In one mode in which a parallel file system can operate, a processor node accesses a single file that is distributed over many I/O nodes by the operating system. (On the paragon, each I/O node is connected to one RAID array. On the nCUBE, each I/O node is connected to a single disk.) If there are more I/O nodes than compute nodes, this may result in improved disk–to–memory transfer rates compared to rates from a single disk to a single processor. However, if there are more compute nodes than I/O nodes (the usual case in massively parallel computers), then the PFS reduces transfer rates due to increased system overhead and conflicts between processors demanding I/O service.

The second mode in which a PFS can operate is using global files, i.e., each file is shared by all processors and is distributed across all I/O nodes. This mode shares both the advantages and the drawbacks of the shared memory paradigm in a massively parallel computing environment.

In a PSS system, each processor has its own logical disk, and the data of a processor's disk is treated similarly to the data in its local memory: the processor will have sole control of this data. Any sharing of data is done by explicit message passing, and thus the PSS system strictly adheres to the distributed memory paradigm. The programming required to make effective use of PSS is more complicated than that required for PFS or for a virtual memory system. However, we expect that anyone already programming in distributed memory environment will be able to use PSS easily and effectively.

The important feature of PSS is that the logical disks can be treated as local secondary storage, and this enables the programmer to control data format and locality. The programmer can thus plan the data format to match the computational requirements and plan the overlap of I/O and computations to match interprocessor communications. The programmer can also plan the data format and I/O to maximize data reuse.

4 Algorithmic issues

In the previous section, we advocated a PSS system for I/O, which conforms to a distributed memory model of programming. In this model, the programmer has the flexibility/responsibility to extract performance from the I/O subsystem by algorithmic modifications. These include changing the data format on the disk to give more efficient access patterns, or combinbing subproblems and redistributing loops to reduce the amount of I/O or communications required. In this section, we discuss several algorithmic issues in the context of an out–of–core LU factorization code.

4.1 I/O complexity

One of the first steps in writing parallel algorithms is to understand how much I/O and interprocessor communication is required by the algorithm and the tradeoffs between ease of programming and computational speed and I/O and communication. This type of analysis often falls under the heading I/O complexity.

An upper bound on the I/O required for LU factorization (with or without pivoting) can be derived by writing the algorithm recursively. We denote the LU factorization of a matrix A by $[L, U] = \mathrm{LU}\,(A)$, where L and U are the lower and upper triangular factors respectively. We also subdivide the matrices A, L and

U into four submatrices and denote these submatrices with subscripts. The LU factorization algorithm can then be written as follows [4].

$$\begin{aligned}
[L_{1,1},\, U_{1,1}] &= \mathrm{LU}\,(A_{1,1}) \\
U_{1,2} &= L_{1,1}^{-1} A_{1,2} \\
L_{1,2} &= 0 \\
U_{2,1} &= 0 \\
L_{2,1} &= A_{2,1} U_{1,1}^{-1} \\
[L_{2,2},\, U_{2,2}] &= \mathrm{LU}\,(A_{2,2} - L_{2,1} U_{1,2})\,.
\end{aligned}$$

If we let n denote the size of the matrix A, M the size of memory and B the size of an I/O request across all processors, then the I/O complexities to compute the products $L_{1,1}^{-1}A_{1,2}$, $A_{2,1}U_{1,1}^{-1}$, and $L_{2,1}U_{1,2}$ are $O(n^3/B\sqrt{M})$ [2, 6, 8]. We can now express the number of I/O operations to factor an $n \times n$ matrix, $T_{LU}(n)$, by the recursion

$$\begin{aligned}
T_{LU}(n) &= 2T_{LU}\left(\frac{n}{2}\right) + O\left(\frac{n^3}{B\sqrt{M}}\right) \\
&= O\left(\frac{n^3}{B\sqrt{M}}\right).
\end{aligned}$$

We note that the upper bound for LU factorization is the same as that for matrix multiplication. We also note that pivoting does not change the upper bound, which can be explained by observing that the complexity of rearranging the rows of a matrix is only $O\left(n^2/B\right)$. From a practical standpoint, pivoting requires a finer subdividing of the matrix so that all the submatrices holding any particular column can be held in memory at the same time.

Because the recursive algorithm can be difficult to implement, we describe a simpler algorithm and discuss its complexity. We begin by dividing the matrix A into b column blocks of size $n \times k$, where $nk = O(M)$ and denoting these blocks by $A_i,\ i = 1, \ldots, b$. Denoting the corresponding components of L and U by L_i and $U_i,\ i = 1, \ldots, b$, we can write LU factorization as follows.

```
for $i = 1, \ldots, b$
    for $j = 1, \ldots, i-1$
        update $A_i$ with $L_j$
    end for
    compute $L_i$ and $U_i$
end for.
```

The high order I/O terms in this algorithm arise from the repeated reading of the L_i. In particular L_i has approximately $(nk - (i - 1/2)k^2)$ entries and must be read $b - i$ times. Summing this up for $i = 1, \ldots, b$, we see that $O(n^4/M)$ entries must be read yielding an I/O complexity of $O(n^4/MB)$.

The complexity of the second algorithm differs form that of the first algorithm by a factor of $n/\sqrt{M}$. A partial solution to this problem is to overlap I/O with computation. In particular, we note that the read of L_{j+1} can be accomplished while L_j is being used for computations. This is, however, only a partial solution; by providing buffer space for I/O, we reduce the memory available for computations by a factor of 2, thereby increasing the amount of I/O. Also, the amount of I/O is greater than the amount of compuations by $O(n)$ so that for very large matrices not all the I/O can be hidden. (We remark here that for all the experiments presented later in the paper, the I/O time was dominated by the computation time.)

4.2 Numerical stability

In the previous subsection, we discussed the use of I/O complexity in choosing an algorithm; however, the primary concern of the programmer must be the stability and correctness of the algorithm being implemented. For example, even though column pivoting is required for the stability of LU factorization for general matrices, many early versions of out–of–core LU factorization incorporated no pivoting or limited pivoting to allow operations on large square blocks that did not contain entire columns. This enabled more efficient use of BLAS 3 routines and reduced the amount of I/O required (even though, as we have shown, the overall complexity cannot be reduced by this technique). While this may be fine for some matrices, the disadvantage is that the resulting codes could not be considered general purpose; the results depended not only on the matrix, but also on the number and configuration of the processors.

4.3 Data format

Once the algorithm and the decomposition of the data to processors and disks have been chosen, the format of the data must be chosen. For example, if a matrix is needed by rows, it does not make sense to store it on disk by columns. Similarly, if a matrix is needed in blocks, as in the recursive algorithms described above, it does not make sense to store it by either rows or columns. Several reports have examined this issue and found that pre– or post–permutations of data lead to a substantially reduced running time for many computations [3].

Sometimes a single computation will require data in different formats for different subcomputations. At other times, the input or output interface may require data in a different format from that which is optimal for the computation. Thus the ability to convert between formats can be important. Several authors have noted that using the I/O system alone to perform these conversions can be quite expensive. Typically data which is contiguous in large blocks in one format is scattered in another. Since disk rates are much lower when servicing scattered small requests, the I/O for the scattered format will suffer. A better alternative is to always read and write data to and from the disks in the disks' current large block format. When data is needed in a new format, the interconnection network can be used to reorder the data, and then write to the disks in the new format [2].

In our most efficient implementation of column–oriented LU factorization, we use two formats. The first is used for the unfactored blocks of the matrix where the matrix is stored by columns. The second is used to store intermediate results and the final (factored) matrix. In this the column blocks are permuted so that the lower triangular entries are stored contiguously and are followed by the upper triangular entries. Because the lower triangular portion must be read into memory repeatedly, this results in less bookkeeping by the program and larger blocks of data transferred.

5 Libraries

For scalability reasons among others, we have advocated a PSS system for out–of–core algorithms, which, as we have observed, imposes an additional burden on the programmer. This burden can be relieved somewhat by the use of libraries.

One of the primary tasks of such a library must be to convert data between various formats. One example of this might be to convert a matrix stored in row format on 1024 virtual disks to one stored in block format on 64 virtual disks. Another example would be to convert the same matrix to a column format on one virtual disk, where the virtual disk is distributed across many physical disk in a PFS format. Many (if not most) change of format changes can be written as bit–permute–complement (BPC) or bit–matrix–multiply–complement (BMMC) transformations. These have been examined in detail in [2] and under other names by several other authors.

Another task of such a library would be to transfer to or from destinations other than disk. For example, the destination might be main processor memory, a HiPPI channel or a graphics frame buffer.

Finally, a useful addition to such a library would be memory mapping services. This would relieve the programmer of tasks such as the calculation of offsets for data transfer and support other services such as caching.

6 Implementations

The test algorithm for our I/O work has been the column–oriented version of LU factorization. As discussed in earlier sections, there is a large amount of I/O; however, this I/O can be hidden by computations for "small" matrices. (Of course, the exact meaning of "small" depends on the relative speeds of computations and I/O for a given machine.) We also use partial pivoting to ensure numerical stability, and permute a column block of the matrix after factoring to make the lower and upper triangular portions of the matrix contiguous within each processor's secondary storage.

We have implemented this algorithm on both the Intel Paragon and the nCUBE 2. The nCUBE 2 at Sandia is a 1,024 node machine. Each node has 4 Mbytes of memory and is capable of 2.1 double precision Mflops/second using the BLAS library. The disk system consists of 16, one Gbyte disks, each with its own SCSI controller.

The Intel Paragon at Sandia has 1840 compute nodes, each with two i860 processors, one for computation and one for comunication. 528 nodes have 32 Mbytes of memory and the remaining nodes have 16 Mbytes of memory. Each node is capable of 45.9 double precision Mflops/second using the BLAS DGEMM routine. There are 48 I/O nodes available, each with a RAID controller and five, one Gbyte disks.

For each machine we have developed versions that run under the vendor–supplied operating systems and versions that run under PUMA, an operating system developed jointly by Sandia National laboratories and the University of New Mexico [7], which implements PSS. The results given later in this section are taken from the nCUBE/PUMA version of the code. The status of the other implementations is as follows.

- nCUBE/Vertex. The nCUBE–supplied software that runs on the disk nodes cannot support the high volume of I/O required by the algorithm. For small problems running on 16 nodes, the compuational rates are similar to those of the nCUBE/PUMA version, while the I/O rates are about half those achievable using the PSS.

- Paragon/PUMA. The I/O support routines in PUMA have not been completed, and we are not able to overlap compuatation and I/O. We typically achieve computational rates in excess of 40 Mflops/node for large problems.

- Paragon/OSF. Due to large memory requirements by the operating system and communication buffer requirements, we have not been able to factor large matrices.

Ideally, we would present experiments that make use of the entire capacity of the machine. Unfortunately, due to the cubic growth of computation time, a single run of the largest matrices requires several hours of computer time. Therefore, we instead present two medium sized runs to demonstrate the ability to overlap I/O with computation and several small sized runs to highlight the dependency on available memory and on the number of processors used.

Table 1 shows the results of running our LU factorization algorithm for a $10,000 \times 10,000$ matrix on 64 processors varying the amount of memory available to the algorithm. In the first run, each block of columns could be made large enough to cover the matrix with 14 block, while in the second run, half the memory was not used thereby halving the potential size of a block and doubling the number of blocks necessary. The increase in the number of blocks almost doubled the amount of I/O done. The last column records the amount of time spent doing I/O that could not be overlapped with computation, which we note is almost constant as predicted by the complexity results.

The increase in total time is almost entirely due to increased interprocessor communication. The amount of communication required is $O(n^2 b\sqrt{p})$, where b is the number of column blocks, and p is the number of processors. Thus the memory size is important in that it defines the grain size for the computation, but not because it affects the amount of visible I/O.

Table 2 shows the scaled efficiency for LU factorization. We note that there is a substantial drop in scaled efficiency in going from one to four processors, but it then remains almost constant up to 64 processors. This can be explained by the physical configuration of the nCUBE, which determines how the PUMA OS assigns disks to processors in the PSS system. There are sixteen disks, each with eight connections to the lower 512 processors of the nCUBE. Thus, there is one connection to a disk for each cube of four processors, and PUMA assigns these four processors to the same disk to minimize traffic through the interprocessor communication network.

Table 3 shows the dependence of the total run time and total I/O on both the number of processors and the memory used for a $2,048 \times 2,048$ matrix. The non–overlapped I/O time is not shown because the small size of the matrix allowed effective caching of data by the disk software in some cases. (Again, the small size was chosen to enable us to do a larger number of runs.) The results again show that the total I/O is inversely proportional to the amount of memory available to the program. The increase in total time, however, is the result of increased interprocessor communication. The data in Table 2 does show that the total I/O is almost independent of the number of processors. Thus the algorithm scales well to large numbers of processors.

7 Conclusions

In this paper we have discussed several aspects of "out–of–core" programming on parallel machines. This is part of the larger problem of moving data into and out of these computers and is characterized by repeated reformatting and transfer of data to and from secondary storage. In particular, we discussed the advantages and disadvantages of several paradigms for disk usage and made the case that an efficient partitioned secondary storage (PSS) is necessary for high performance scientific computation.

We also discussed many of the programming issues which arise when using a disk system for temporary storage. These included data partitioning, data format, I/O complexity, numerical stability and the use of libraries. In each of these discussions, we used LU factorization as an example and in the end showed that the use of PSS (the "block server" facility in the PUMA operating system) leads to a very efficient out–of–core LU code. We also showed that an important feature of any I/O library or PSS system is the ability to do background I/O. This enabled us to substantially reduce the "visible" I/O time in LU factorization.

Even though we used an LU factorization code to demonstrate the ideas of parallel I/O, most of the discussions in this paper apply equally well to any code that requires repeated access to large blocks of data. (The discussion of I/O complexity, of course, is specific to LU.)

Acknowledgements

This work was supported in part by the U.S. Department of Energy and was performed at Sandia Na-

Table 1: Times to factor a double precision $10,000 \times 10,000$ matrix using 64 processors on the nCUBE 2

number of column blocks	column–block size (bytes/proc)	memory used	total I/O (Mbytes)	total time (sec)	"visible" I/O time (sec)
14	893,214	94%	4,924	6,320	85
27	463,148	49%	8,869	6,574	87

Table 2: Scaled efficiency for LU factorization on an nCUBE 2

p	n	total I/O (Mbytes)	total time (sec)	"visible" I/O time (sec)	scaled efficiency
1	1,250	77	742	15	1.00
4	2,500	308	1,585	54	0.94
16	5,000	1,231	3,146	55	0.94
64	10,000	4,924	6,320	59	0.94

Table 3: Times to factor a double precision $2,048 \times 2,048$ matrix on the nCUBE 2

p	number of column blocks	column block size (bytes/proc)	memory used	total I/O (Mbytes)	total time (sec)
16	4	524,800	55%	71	231
16	8	242,400	26%	132	254
32	2	525,312	55%	34	117
32	4	262,656	28%	71	126
32	8	131,328	14%	132	141
64	2	262,656	28%	34	67
64	4	131,328	14%	71	77
64	8	65,664	7%	132	99

tional Laboratories operated for the U.S. Department of Energy under contract No. DE-AC04-94Al85000

References

[1] C. M. Burns, R. H. Kuhn, and E. J. Werme, “Low copy message passing on the alliant CAMPUS/800”, in *Proceedings of Supercomputer '92, 1992,* pp. 760–769.

[2] T. H. Cormen, “Virtual Memory for Data-Parallel Computing”, PhD thesis, MIT, 1993.

[3] J. M. del Rosario, R. Bordawekar, and A. Choudhary, “Improving parallel I/O performance using a two-phase access strategy”, Tech. Report SCCS-406, Northeast Parallel Architectures Center, 1993.

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 2nd ed., 1989.

[5] B. A. Hendrickson and D. E. Womble, “The torus–wrap mapping for dense matrix calculations on massively parallel computers”, *SIAM J. Sci. Comput.*, (to appear).

[6] J.-W. Hong and H. T. Kung, “I/O complexity: The red–blue pebble game”, in *Proceedings of the Symposium on the Theory of Computing, 1981*, pp. 326–332.

[7] A. B. Maccabe and S. R. Wheat, “Message passing in SUNMOS”. Overview of the Sandia/University of New Mexico OS, now called PUMA., Jan. 1993.

[8] J. S. Vitter and E. A. M. Shriver, “Algorithms for parallel memory I: Two–level memories”, Tech. Report CS–92–04, Brown University, 1992.